Computational Complexity

Prof. Michele Amoretti

High Performance Computing 2022/2023

Computational Complexity

The **computational complexity**, or simply **complexity** of an **algorithm** is the amount of resources required for running it.

The computational complexity of a **problem** is the minimum of the complexities of all possible algorithms for this problem (including the unknown algorithms).

A **complexity class** can be thought of as a collection of computational problems requiring O(f(n)) resources of an abstract machine for being solved, where n is the size of the input.

Notation for Complexity

We say g(n) = **O(f(n))** if there exist constants a,b such that g(n) < a f(n) + b, for all n.

We say g(n) = **Ω(f(n))** if there exist a > 0, b such that g(n) > a f(n) - b, for all n.

We say g(n) = **Θ(f(n))** if g(n) = O(f(n)) and g(n) = Ω(f(n)).

Decision Problems and Languages

A **decision problem** is a YES-or-NO question about the input values.

Solving a decision problem is equivalent to **deciding the language L={x : f(x)=1}**, where f() is the function that defines the problem.

R = class of all languages decidable on some Turing machine.

All the problems that belong to **R** are defined by computable functions f().

The halting problem is not in **R**.

Most decision problems are not in **R**.

Upper Bounds vs Lower Bounds

For a lot of problems, we don't know the problem complexity, but we have bounds on it.

We know the problem complexity falls between an upper bound and a lower bound.

Discovery of better algorithms reduces the upper bound on the worst-case computational complexity.

- Suppose we have problem P and someone finds algorithm A of running time $O(n^3)$.
- So we know the time complexity of the problem cannot be higher than $O(n^3)$.
- Later, someone discovers a better algorithm, whose running time is $O(n^2)$.
- So the time complexity of the problem cannot be higher than $O(n^2)$.

Upper Bounds vs Lower Bounds

Proving statements about lower bounds can be hard!

We have to prove something that will hold true for all possible algorithms that solve the problem (not just algorithms that exist but ones that could be discovered in the future).

Each better proof brings the lower bound upwards.

– Suppose we prove that the problem simply cannot be solved in less than linear time, O(n). In other words, no algorithm could possibly exist that would solve this problem in less than O(n) time.
– Note we are not coming up with algorithms (unlike the story with the upper bound). We prove that no such algorithm could exist so the time complexity of the problem is Ω(n).
– Later, with more sophisticated techniques, we find a proof that the problem cannot be solved in less than O(nlogn) time, i.e., we prove no such algorithm could exist so the time complexity of the problem is Ω(nlogn).

Time vs Space Complexity

Time complexity vs space complexity:

TIME(f(n)) is the class of languages that are decidable by an O(f(n)) time deterministic Turing machine.

SPACE(f(n)) is the class of languages that are decidable by an O(f(n)) space deterministic Turing machine.

A **time-efficient algorithm** is one that **runs in polynomial time**, i.e., $O(n^k)$, in the size of the problem solved.

An algorithm that requires super-polynomial (typically exponential) time to solve a problem is not time-efficient.

Deterministic TM = classical computer.

Complexity Relationships Among Models

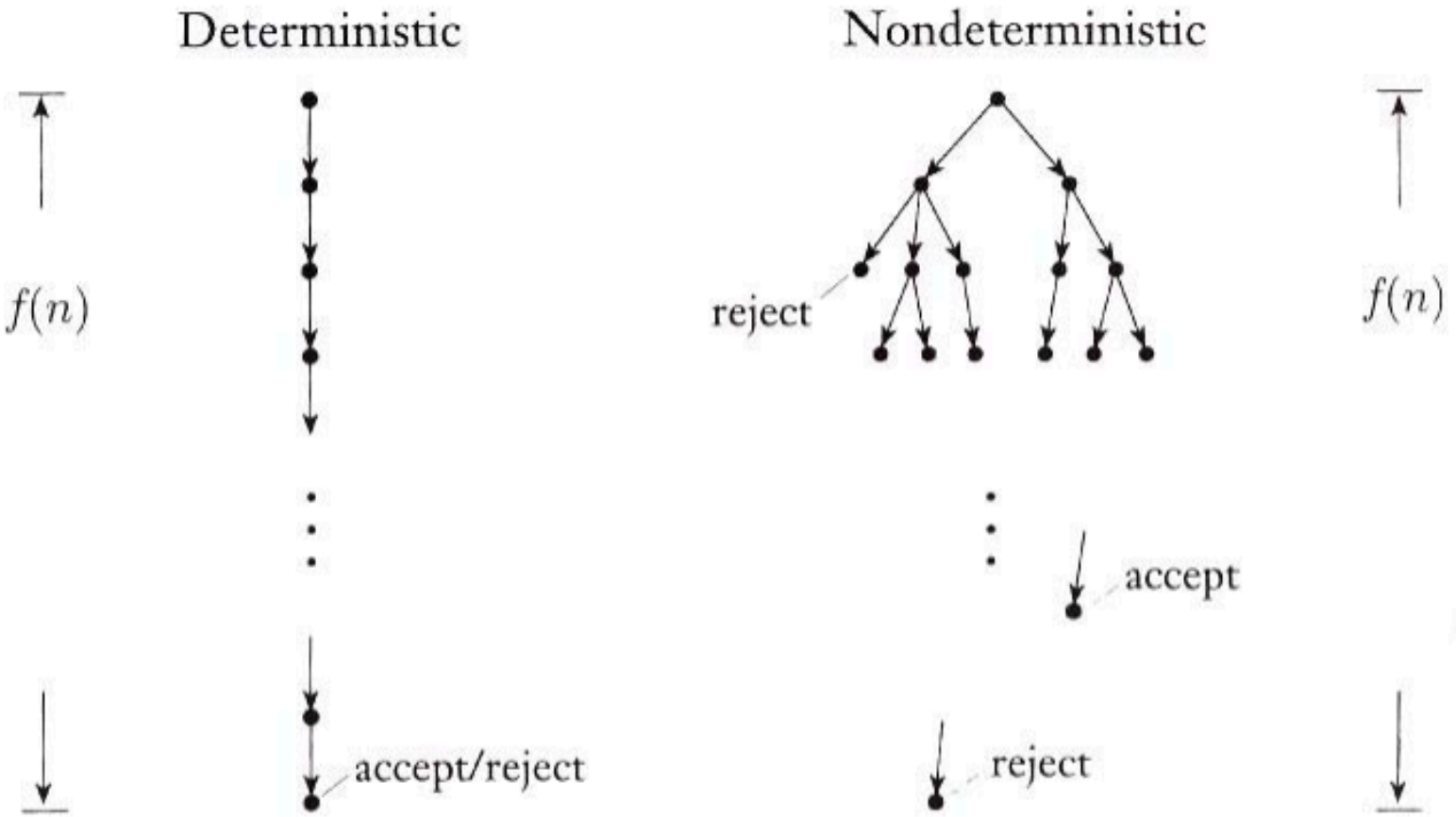
The choice of computational model can affect the time complexity of languages.

Theorem - Let f(n) be a function, where f(n) ≥ n. Then every f(n) time multitape Turing machine has an equivalent $O(f^2(n))$ time single-tape Turing machine.

Theorem - Let f(n) be a function, where f(n) ≥ n. Then every f(n) time nondeterministic single-tape Turing machine has an equivalent $2^{O(f(n))}$ time deterministic single-tape Turing machine.

Complexity Relationships Among Models

Measuring deterministic and nondeterministic time:



P and EXP

P is the class of languages that can be decided quickly (i.e., in polynomial time) on a classical computer.

P is the union, over all positive integers k, of **TIME(n^k)**. In other words, it is the class of languages that are decidable in polynomial time on a deterministic Turing machine.

Examples:

- sorting the elements of a list in a certain order
worst-case complexity $O(n^2)$, average time complexity $O(n \log n)$

- finding the greatest common divisor (GCD) of two integers
solved in polynomial time by the Euclidean algorithm

EXP is the union, over all positive integers k, of **TIME(2^{n^k})**.

P/*f(n)*

P/f(n) is the class of languages that are decidable by a polynomial-time deterministic TM, given the input w of length n and an advice string c of length f(n).

The advice string is just an extra input.

P/poly --> the length of c is polynomial in n

P/poly is important for several reasons.
For example, in cryptography, security is often defined against **P/poly** adversaries.

NC

NC consists of those languages that can be decided in polylogarithmic time on a parallel computer with a polynomial number of processors.

More precisely: find constants k and m such that the language is decided in time $O(\log^m n)$ using $O(n^k)$ processors.

NC is a subset of **P**, believed to be strict (i.e., there are tractable problems inherently sequential).

Examples:

- integer addition, multiplication and division
- matrix multiplication, inverse, determinant, rank
- finding a maximal matching

NP

A **verifier** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w,c \rangle \text{ for some string } c\}$$

We measure the time of a verifier only in terms of the length of w.

NP is the class of languages that have polynomial time verifiers.

This means that if someone gives us an instance of the problem and a certificate (sometimes called a proof) to the answer being YES, we can check that it is correct in polynomial time.

NTIME(t(n)) is the class of languages that are decidable by an O(t(n)) time nondeterministic Turing machine.

NP is the union, over all positive integers k, of **NTIME(n^k)**.

coNP

coNP contains the languages that are complement of languages in **NP**.

This means that if someone gives us an instance of the problem and a certificate to the answer being NO, we can check that it is correct in polynomial time.

We don't know whether **coNP** is different from **NP**.

Integer Factorization

Given an integer N and an integer M with $1 \leq M \leq N$, does N have a prime factor d, with $1 < d < M$?

It is in **NP**, bacause a prime factor d, with $1 < d < M$, for N serves as a witness of a YES instance. We can check the witness in polynomial time by performing the division N/d.

It is in **coNP**, because a prime factorization of N with no factors < M serves as a witness of a NO instance (recall that prime factorizations are unique).

The best classical algorithm for factoring an integer is the **number field sieve**, whose time complexity is super-polynomial in n = logN.

http://mathworld.wolfram.com/NumberFieldSieve.html

Polynomial-time Reducibility

Polynomial-time reducibility allows us to classify problems according to relative difficulty.

Language A is **polynomial-time reducible** to language B, written $\mathbf{A \leq_p B}$, if there is a polynomial-time computable function f, where for every w,

$$w \in A \Leftrightarrow f(w) \in B.$$

The function f is called the **polynomial-time reduction** of A to B.

If $A \leq_p B$ then A cannot be harder than B, because whenever an efficient algorithm exists for B, one exists for A as well. Conversely, if no efficient algorithm exists for A, none exists for B either.

Example: finding the median reduces to sorting

If $A \leq_p B$ and $B \leq_p A$ then A and B are **computationally equivalent** ($\mathbf{A \equiv_p B}$).

Hardness and Completeness

Problem B is called ***C*-Hard** if for all problems A in ***C*** we have $A \leq_p B$.

If we further have that B is in ***C***, then B is called ***C*-Complete**.

In general, ***C*** and ***C*-Complete** are different classes, but we cannot say that their intersection is empty for all ***C***.

C*-Complete** problems are the hardest problems in the class ***C. If we manage to solve one of them efficiently, we have done so automatically for all other problems in ***C***. Instead, if we manage to solve efficiently a problem in ***C*** that is not ***C*-Complete**, we are still unable to solve efficiently the problems that are ***C*-Complete**.

Problems in ***C*-Hard** are at least as hard as the hardest problems in ***C***, (i.e., the ***C*-Complete** problems).

Tractable vs. Intractable Problems

Tractable problems can be solved in polynomial time, i.e., are in **P**.

Intractable problems seem to require exponential time.

If $A \leq_p B$ and B is tractable, then so is A. Solve B in polynomial time, then use polynomial-time reduction to solve A.

If $A \leq_p B$ and A is intractable, then so is B. Suppose B could be solved in polynomial time. Then so could A, through polynomial-time reduction. We have a contradiction, therefore B is intractable.

NP-Hard

Language B is **NP-Hard** if any language A in **NP** is polynomial-time reducible to B.

I.e., **NP-Hard** problems are at least as hard as any other problem in **NP**.

The halting problem is not in **NP**, because it is not decidable in a finite number of operations. However, it is **NP-Hard**.

NP-Complete

Language B is **NP-Complete** if it satisfies two conditions:

1. B is in **NP**
2. B is **NP-Hard**

All **NP-Complete** problems are in a sense "the same problem", as any **NP-Complete** problem is polynomial-time reducible to any other **NP-complete** problem in polynomial time.

NP-Complete

Many problems of practical interest are **NP-Complete**:

- SAT (Cook-Levin theorem)
- 3SAT
- CircuitSAT
- Map Coloring
- 3-Partition
- Bin Packing
- Traveling Salesman Problem (TSP)

Integer factorization is in **NP** and **coNP**. It is suspected to be outside the **P, NP-Complete** and **coNP-Complete** complexity classes.

PSPACE

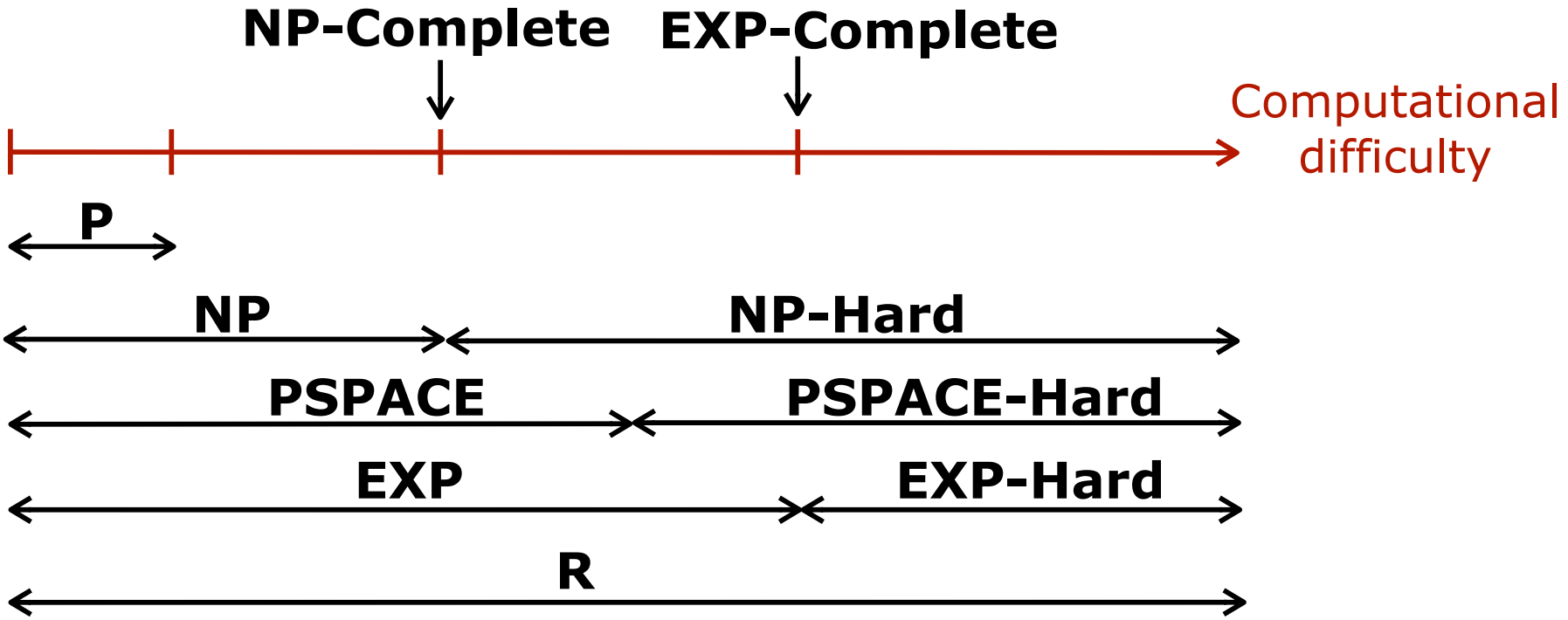
PSPACE is the class of languages that are decidable in polynomial space on a deterministic Turing machine. In other words, **PSPACE** is the union, over all positive integers k, of **SPACE(n^k)**.

$\subseteq$**EXP**: the total number of configurations of the TM is at most $2^{q(n)}$ for some polynomial q(n)

$\supseteq$**NP**: cycle through all certificates, storing them one by one in the work tape of the TM (assuming a three-tape TM)

Examples:
- Tic-Tac-Toe
- Reversi

Comparing classes by computational difficulty



P vs. NP

Clearly, **P** is a subset of **NP** (if a problem can be solved in polynomial time, it can be also verified in polynomial time on a deterministic TM).

However, it is not so clear whether or not there are problems in **NP** that are not in **P**.

P vs. **NP** is probably the most important unsolved problem in theoretical computer science.

If any one of the **NP-Hard** problems was to be solved quickly, then all **NP** problems could be solved quickly (which would mean that **P=NP**).
However, we usually focus on **NP-Complete** problems, because of their practical interest, and we try to find an efficient solution for any one of them.
On the other hand, proving that any one of the **NP-Complete** problems does not have an efficient solution, would mean that **P!=NP**.

Oracles

An **oracle** is a black box able to solve a decision problem in O(1) time (i.e., constant time).

$\mathbf{P}^{O}$ is the class of languages that are decidable in polynomial time, given oracle access to problem O.

Example:

Let $\overline{SAT}$ denote the language of unsatisfiable formulae.

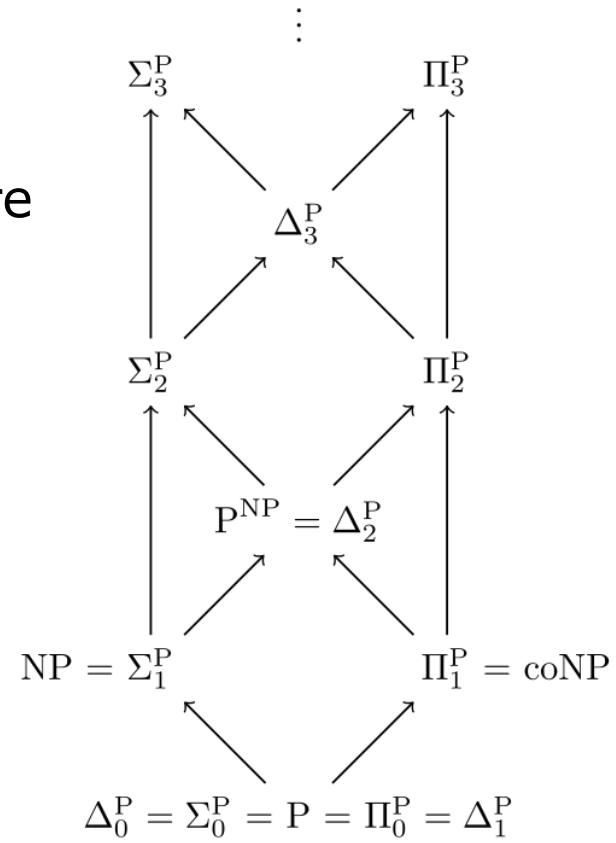
Then $\overline{SAT}$ belongs to $\mathbf{P}^{SAT}$.

PH

The polynomial hierarchy is $\{\Delta_i^P, \Sigma_i^P, \Pi_i^P; i \geq 0\}$, where

$\Delta_0^P = \Sigma_0^P = \Pi_0^P = P$ and, for i>0:

- $\Delta_i^P = P$ with Σ_{i-1}^P oracle
- $\Sigma_i^P = NP$ with Σ_{i-1}^P oracle
- $\Pi_i^P = coNP$ with Σ_{i-1}^P oracle



Examples:

$\Delta_1^P = P^P = P$

$\Delta_2^P = P^{NP}$

$\Sigma_1^P = NP$

$\Pi_1^P = coNP$

PH is the union of all classes in the polynomial hierarchy.

If $\Sigma_k^P = \Sigma_{k+1}^P$ or $\Sigma_k^P = \Pi_k^P$ then the polynomial hierarchy collapses to level k. If **P** = **NP** then the hierarchy collapses completely.

#P

#P is the set of the counting problems associated with the decision problems in **NP**.

Rather than "are there any", we ask "how many" solutions.

Clearly, a problem in **#P** must be at least as hard as the corresponding **NP** problem.

A polynomial time deterministic Turing machine with a **#P** oracle can solve all problems in **PH**.

Examples:
- Matrix permanent (#P-Hard, in general; #P-Complete if A is binary)

$$\operatorname{perm}(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{i,\sigma(i)}$$

Randomized algorithms

In the mid 1970s, **randomized algorithms** were introduced, suggesting that some problems that have no efficient solution on a deterministic TM, may have an efficient solution on a **probabilistic TM**, which is a type of nondeterministic TM (each step has two legal next moves and the probability of a branch is 2^{-k}, where k is the number of steps that occur on the branch).

Example: Solovay-Strassen test for primality

In 2002, Agrawal, Kayal and Saxena later showed that primality-testing can be performed in polynomial time on a deterministic TM.

M. Agrawal, N. Kayal, N. Saxena, *PRIMES is in P*, Annals of Mathematics 160 (2): 781–793 (2004)

The general project of taking randomized algorithms and converting them to deterministic ones is called **derandomization**.

BPP

BPP (Bounded-error Probabilistic Polynomial-time) is the class of languages that are decided in polynomial time by a probabilistic Turing machine, with an error probability of 1/3.

The relationship between **BPP** and **NP** is unknown: it is not known whether **BPP** is a subset of **NP**, **NP** is a subset of **BPP**, or neither.

Instead, there is an increasing evidence that **P=BPP**.

Example:
- determining whether a given multivariate polynomial has roots

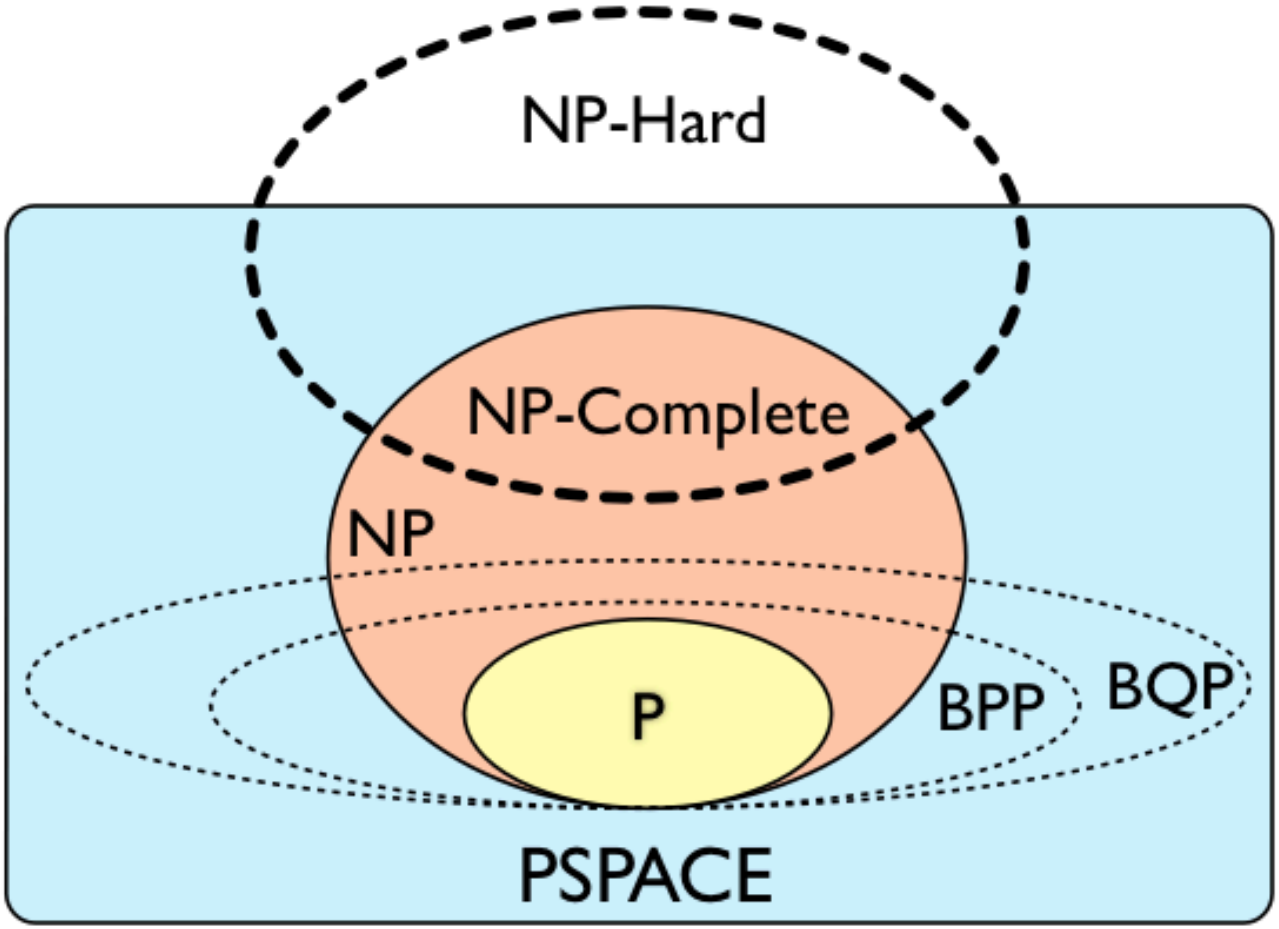
BQP

BQP (Bounded-error Quantum Polynomial-time) is the class of languages that are decided in polynomial time by a universal quantum computer, with an error probability of 1/3.

It is known that **BQP** includes **BPP**, but where **BQP** fits with respect to **P**, **NP** and **PSPACE** is yet unknown. What is known is that quantum computers can solve all the problems in **P** efficiently, but that there are no problems outside **PSPACE** that they can solve efficiently. Thus, **BQP** lies somewhere between **P** and **PSPACE**.

Examples:
- Integer factorization ---> Shor's algorithm has $O(n^3)$ time complexity
- Discrete logarithm
- Simulation of quantum systems

Venn diagram for complexity classes



References

• M. Sipser, *Introduction to the Theory of Computation*, 3rd ed., Cengage Learning, 2013

• E. Demaine, *Algorithmic Lower Bounds: Fun with Hardness Proofs*, MIT Course 6.890, 2014

• M. Nielsen, I. Chuang, *Quantum Computation and Quantum Information*, Cambridge University Press, 2011

• S. Aaronson, *Quantum Computing Since Democritus*, Cambridge University Press, 2013

• R. R. Williams, *Some Estimated Likelihoods For Computational Complexity*, Invited paper in the Springer LNCS 10000 volume, 2018

• **https://complexityzoo.net/Complexity_Zoo**